Multiprogram Throughput Metrics: A Systematic Approach

Stijn Eyerman, Pierre Michaud, Wouter Rogiest

Multi-Program Throughput Metrics: a Systematic Approach

STIJN EYERMAN, Ghent University
PIERRE MICHAUD, INRIA Rennes
WOUTER ROGIEST, Ghent University

Running multiple programs on a processor aims at increasing the throughput of that processor. However, defining meaningful throughput metrics in a simulation environment is not as straightforward as reporting execution time. This has led to an ongoing debate on what forms a meaningful throughput metric for multi-program workloads. We present a method to construct throughput metrics in a systematic way: we start by expressing assumptions on job size, job distribution, scheduling, etc., that together define a theoretical throughput experiment. The throughput metric is then the average throughput of this experiment. Different assumptions lead to different metrics, so one should be aware of these assumptions when making conclusions based on results using a specific metric.

Throughput metrics should always be defined from explicit assumptions, because this leads to a better understanding of the implications and limits of the results obtained with that metric. We elaborate multiple metrics based on different assumptions. In particular, we identify the assumptions that lead to the commonly used weighted speedup and harmonic mean of speedups. Our study clarifies that they are actual throughput metrics, which was recently questioned. We also propose some new throughput metrics, which cannot always be expressed as a closed formula. We use real experimental data to characterize metrics and show how they relate to each other.

1. INTRODUCTION

Metrics are at the foundation of experimental evaluation. Choosing the correct metric is as important as setting up a meaningful experiment. The choice of the metric should reflect the goal of the study as closely as possible. For example, a cache optimization study can report cache miss rate reductions, but if the end goal is to improve performance, execution time reductions are more meaningful. Ideally, a metric should represent an externally observable characteristic of the system (execution time, response time, power consumption, etc.).

Evaluating performance for *single-program* workloads (single- or multi-threaded) is straightforward: the end goal is to reduce the execution time. To avoid giving more weight to longer running benchmarks, rate-based metrics are frequently used: IPC (instructions per cycle) for single-threaded programs and speedup versus single-threaded execution for multi-threaded programs. On *multi-program* workloads, though, the goal is not to decrease the execution time of individual programs, but to increase the throughput of the system. Throughput is less strictly defined than execution time, leading to many possible interpretations. As a result, there is an ongoing debate on what forms a good throughput metric for multi-program workloads in the context of a (micro)architecture study, where we want to compare two or more configurations using simulation [Snavely and Tullsen 2000; Luo et al. 2001; Eyerman and Eeckhout 2008; 2013; Michaud 2013].

Throughput is generally defined as the number of jobs executed per unit of time. The exact throughput number depends on many characteristics of the setup: job arrival times, job behaviors, job sizes, scheduling policy, etc. Ideally, maximum throughput should be measured using a *throughput experiment*: a simulation setup where jobs arrive, are scheduled, executed, and leave the system. The problem is that it is not easy

Stijn Eyerman and Wouter Rogiest are postdoctoral fellows of the Research Foundation – Flanders (FWO-Vlaanderen). Author's addresses: S. Eyerman, ELIS Department, W. Rogiest, TELIN Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; P. Michaud, INRIA, Rennes; email: Stijn.Eyerman@elis.UGent.be, Wouter.Rogiest@telin.UGent.be, Pierre.Michaud@inria.fr.

to set up an actual throughput experiment and it is a challenge to perform sufficient representative experiments, due to the large number of possible settings and due the fact that simulation is very slow, often preventing the complete simulation of realistic jobs.

A common practice in microarchitecture studies using multi-program workloads is to select a sample of *coschedules*, i.e., some benchmark combinations, and simulate each coschedule separately. Typically, the number of benchmarks in a coschedule does not exceed the maximum number of threads that can run in parallel, to avoid modeling context switches and scheduling. In the end, the results of these simulations should be averaged into a throughput number that is close to that of an actual throughput experiment. This averaging is not uniquely defined, because it relies on assumptions that may impact the final throughput.

In this paper, we present a method to construct a meaningful throughput metric without doing an actual throughput experiment. We elaborate some new metrics and give a new meaning to known metrics. Because an absolute throughput number depends on many assumptions, there is no single throughput metric. In fact, a particular study can require constructing a metric that is specific to that study only. Furthermore, different assumptions lead to different metrics, which can result in contradictory conclusions. However, it is important to have a systematic method to construct metrics, such that *we can choose a metric that is consistent with the goal of a study, instead of using an arbitrary metric with an unclear meaning*.

The starting point for constructing a metric is to define a theoretical throughput experiment (TPEX). The theoretical TPEX is defined by assumptions about job sizes, job distribution, scheduling, etc., such that all assumptions completely determine an experiment that could be done in reality (but that would take too much time and effort). The throughput metric is then the throughput measured on this TPEX. This means that a metric is defined by the assumptions of the TPEX and not by a mathematical formula. A mathematical formula can be derived from the assumptions, forming a practical way to calculate throughput. However, we will see that it is not always possible to construct a closed formula for each TPEX.

In this paper, we make the following contributions:

— We introduce the concept of a theoretical throughput experiment to define a multi-program throughput metric.
— We are the first to define fixed workload throughput metrics.
— We provide practical solutions to calculate fixed workload throughput metrics, including a software tool that calculates these metrics given simulated performance data.
— We evaluate the metrics under discussion using simulated performance data, and compare their behavior and implications on selecting optimal designs.
— We discuss how to calculate the average power consumption of a TPEX.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces the theoretical throughput experiment. Section 4 shows how to calculate instantaneous throughput, i.e., the throughput of one coschedule, while Section 5 elaborates average throughput metrics. Section 6 discusses techniques to obtain fixed workload throughput metrics and Section 7 discusses practical considerations for using these metrics in a constrained simulation setup. Section 8 analyzes the impact of choosing a metric on a realistic experiment and Section 9 discusses calculating average power consumption. We finally discuss future work and conclude in Sections 10 and 11.

2. RELATED WORK

Computer architects consider several sorts of workloads for evaluating processor performance. Multi-program workloads represent the typical situation where several independent jobs run concurrently on a processor. Computer architects generally use multi-program workloads to study problems concerning resource sharing, e.g., should we share a resource or replicate it, how do we manage a shared resource, etc.

Multi-program workloads can be studied by running throughput experiments (TPEX). In a typical TPEX, the workload consists of a relatively large number of jobs: when a core becomes free after the job running on it has finished, it can execute another job.

The SPEC organization defines a metric called SPECrate, based on the SPEC CPU benchmarks [Carlton 1995]. SPECrate can be measured by running TPEXs. It assumes *homogeneous* workloads, i.e., multiple copies of the same benchmark. Though homogeneous workloads represent an important case, not all workloads are homogeneous. Multi-program workloads are often *heterogeneous*, consisting of two or more different sorts of jobs. SPECrate is not sufficient to understand the effects of workload heterogeneity.

A few computer architecture studies have done some TPEX simulations (e.g., [Snavely and Tullsen 2000; Parekh et al. 2000; Kumar et al. 2004; Najaf-abadi and Rotenberg 2009; Eyerman and Eeckhout 2010]). However, microarchitecture simulators are slow and make it difficult to simulate TPEXs. This is why the use of TPEXs in computer architecture studies remains infrequent.

In the 1990's, the first studies on Simultaneous Multithreading (SMT) processors used total IPC (instructions executed per cycle) to compare different SMT configurations [Tullsen et al. 1995; Tullsen et al. 1996]. However, at that time, there was a growing concern that total IPC was not the right metric, as it tends to advantage high-IPC threads over low-IPC ones. Therefore researchers looked for a more faithful metric.

Several researchers independently proposed what is now usually called *weighted speedup* [Snavely and Tullsen 2000; Parekh et al. 2000; Sazeides and Juan 2001]. Weighted speedup is the sum of the speedups of all the threads running concurrently, speedup being defined relative to isolated execution. Weighted speedup is a throughput metric, it does not express fairness. Fairness means that shared processor resources should be fairly divided among competing threads, so that each thread makes forward progress at a reasonable rate, where different definitions of what "reasonable" means lead to different fairness metrics [Luo et al. 2001; Vandierendonck and Seznec 2011]. Luo et al. proposed to mix throughput and fairness in a single metric, the harmonic mean of speedups (H-mean) [Luo et al. 2001].

Weighted speedup and H-mean speedup are the most frequently used multi-program performance metrics in computer architecture studies. Many studies use both metrics simultaneously.

Still, some questions were unanswered: Do these metrics provide an objective measure of performance? Is a higher weighted/H-mean speedup a concrete improvement for the end user? Eyerman and Eeckhout searched for a relation between weighted/H-mean speedup and some tangible system-level quantities, namely job throughput and turnaround time [Eyerman and Eeckhout 2008]. Michaud argued that weighted speedup and H-mean speedup are *inconsistent*, in the sense that they make performance depend on the choice of a reference machine [Michaud 2013]. Michaud also proposed H-mean IPC as a new multi-program throughput metric. Eyerman and Eeckhout, as a response to Michaud's paper, restated the case for speedup-based metrics, arguing that speedup, unlike raw IPC, has a system-level meaning [Eyerman and Eeckhout 2013].

Previously introduced throughput metrics, including weighted/H-mean speedup, seek to quantify *instantaneous* throughput, that is, the throughput of one particular *coschedule*. A coschedule is a combination of K jobs, K being the maximum number of threads that can run simultaneously on the processor. Researchers generally form several different coschedules from a set of benchmarks (e.g., SPEC CPU). Several questions arise: How to select the coschedules? Are all coschedules equally important? How to compute an average throughput from the per-coschedule throughputs? In most microarchitecture papers concerned with multi-program workloads, the coschedules are selected "manually" from a set of benchmarks, based on the authors' intuition of the particular problem under study. In a few papers, more systematic methods are used for selecting the coschedules. Van Biesbrouck et al. classify co-phases based on microarchitecture-independent characteristics, and they use clustering to identify a set of representative co-phases [Van Biesbrouck et al. 2007]. Random sampling is used in some studies. Van Craeynest and Eeckhout show that for averaging throughput over the full population of possible coschedules, it may be more accurate to use a large random sample and fast approximate simulation than a small sample of manually chosen coschedules and detailed simulation [Van Craeynest and Eeckhout 2011]. Velásquez et al. show that fast approximate simulations may help select a representative sample of coschedules for detailed simulations [Velásquez et al. 2013].

Concerning the phase behavior of programs, Kihm et al. [Kihm et al. 2005] and Van Biesbrouck et al. [Van Biesbrouck et al. 2006] argue that all the possible offsets between coexecuting jobs should be considered equally likely. They propose a method that gives a per-coschedule average throughput. Another frequently used method considers several representative simulation points per benchmark, and forms coschedules by combining simulation points instead of benchmarks.

When simulating the execution of one coschedule for a fixed time, the work executed by a job depends on the machine's performance, i.e., different machines do not execute the exact same work. However, microarchitects generally prefer to compare different processors on the same work [Sazeides and Juan 2001; Hilton et al. 2009]. To solve this issue, Hilton et al. proposed to fix the work on which two processors are being compared, i.e., each job executes a fixed number of instructions [Hilton et al. 2009]. When a job finishes, the core on which it executed remains idle. To minimize the "tail" effect, i.e., the idle time at the end of the simulation, Hilton et al. propose to dimension job sizes so that all jobs have the same run time when executed alone. Another solution is to re-execute a job or a simulation point as many times as necessary for the tail effect to be negligible [Tuck and Tullsen 2003; Vera et al. 2007]. Another frequently used method is to measure the IPC of each simulation point, but let the execution continue past the simulation point until all simulation points in the coschedule have been fully executed, so that contention effects on shared resources are properly simulated.

Our motivations and contributions. The multi-program performance metrics used in computer architecture studies so far have not been defined from clear and precise assumptions. The main goal of this work is to define multi-program throughput metrics from precise assumptions, our stance being that throughput should ideally be measured from a throughput experiment. We propose a new family of throughput metrics that allow to compare different processors executing the exact same workloads. Because simulating full throughput experiments with slow microarchitecture simulators is difficult, we propose a method for computing average throughput from independent coschedule simulations.

3. THE THEORETICAL THROUGHPUT EXPERIMENT

In this section, we introduce the concept of a theoretical throughput experiment. We also discuss job types to abstract away non-representative characteristics of bench-

Table I. Example experiment

	Reference core IPC	Multicore IPC Co-run with A	Co-run with B
Job type A	2	2	1
Job type B	1	0.5	1

marks, and the difference between the number of benchmarks and the number of jobs in a TPEX.

3.1. Benchmark Versus Job Type Versus Job

Computer architecture researchers heavily rely on benchmarks. These benchmarks are supposed to represent realistic workload behavior in a certain application domain. However, usually no information is available about realistic program sizes and the relative occurrence of (combinations of) programs. The size (e.g., instruction count) of benchmarks, even with so-called reference inputs, is not representative for real jobs. This has already been recognized in other studies, such as the fact that long running benchmarks should not be weighted more in calculating average performance [John 2006].

The behavior of a benchmark, i.e., the way its instructions interact with the hardware, is the only representative part of a benchmark. We therefore define a *job type* as the behavior of a certain (phase of a) benchmark. A *job* is then an instantiation of a job type with a certain job size (number of instructions). We assume a job to be homogeneous, i.e., its behavior is constant, independent of the number of instructions (over a sufficiently large amount of instructions). This might seem unrealistic, because many programs show some phase behavior. On the other hand, just like the instruction count of the full benchmark, the instruction count of phases and the switching point of phases in a benchmark is also not representative. In our framework, phase behavior can be modeled by starting a new job with a different job type. Each phase represents a different job type, and job types can also be weighted according to the probability of a phase to occur (similar to the multiple SimPoint technique [Sherwood et al. 2002]).

3.2. The Throughput Experiment

A typical multi-programmed system has no fixed combination of programs that is constantly running. Jobs come and go, and the throughput of the system is highly dependent on what jobs (co-)execute at which time. Setting up such an experiment is hard, both technically and in terms of representativeness. In a simulated environment, which is common for computer architecture studies, full system simulation is required, the arrival time and execution time of individual jobs needs to be monitored, etc. Furthermore, the number of possible job arrival times, job types and job sizes leads to a gigantic exploration space, making it difficult to find a meaningful set of experiments.

Therefore, computer architecture researchers typically simulate a certain set of benchmark combinations, where each simulation consists of a fixed benchmark combination, or *coschedule*. However, this does not represent a real throughput experiment, where jobs come and go, and where the performance of individual jobs can have an impact on which jobs are coscheduled. Figure 1 illustrates the difference. The top figure represents a real throughput experiment for two cores and two job types (A and B), while the bottom part illustrates the isolated simulations of each coschedule individually. Each of the boxes represents a job: a dark gray box is job type A, light gray is job type B. The horizontal axis is time, and the top box is the job run on core 1, the bottom box is the one on core 2. The height of a box is its IPC. The IPC values for this example setup can be found in Table I. We will use the same example throughout the whole paper to visualize the different metrics.

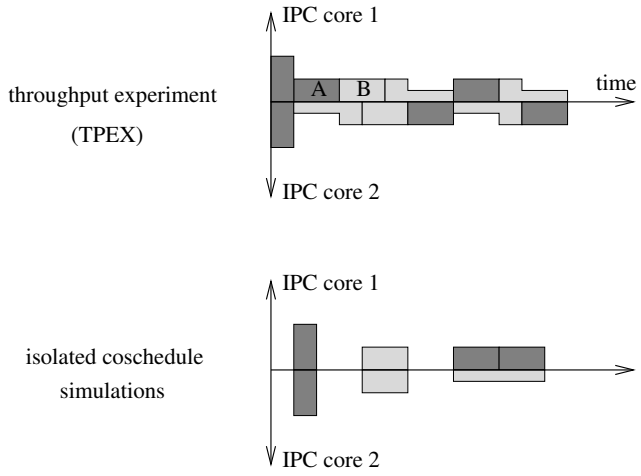

Fig. 1. Throughput experiment vs. isolated coschedule simulations.

From this figure, it is clear that the isolated experiments quantify the impact on performance of co-running jobs, but the impact on the throughput of the real throughput experiment is not immediately clear. For example, a job that is delayed more due to interference with another job will take longer to execute, and therefore the probability of the combinations that are run concurrently is changed. *The goal of this paper is to present a methodology to approximate the average throughput of a TPEX, using performance data from isolated experiments.*

We first define some terms and general assumptions to ease the discussion:

Coschedule. A specific combination of job types that is running concurrently is called a coschedule. In a TPEX, the coschedule changes when a job is done (or scheduled out) and a new one is scheduled. The isolated experiments evaluate one coschedule each.

Job homogeneity. As discussed before, job behavior is assumed to be homogeneous. As a result, the interference between jobs in a specific coschedule is also homogeneous, and the performance per job in a coschedule is constant (but different for different coschedules). The performance of each job in a coschedule equals the performance measured in the isolated experiments. This means that we ignore any transition effects when the coschedule is changed, which is only accurate when coschedules are run long enough.

Instantaneous throughput. Instantaneous throughput is the throughput of one specific coschedule. It is the throughput at a specific moment in time of the TPEX, namely when that coschedule is running.

Average throughput. Average throughput is the total throughput of the full TPEX. It is an average of the instantaneous throughput of all coschedules of that TPEX, weighted by the probability of a coschedule to occur. This is the final throughput metric we want to calculate.

Full load. We assume the processor to be always fully loaded, i.e., the number of running jobs is always equal to the number of cores or thread contexts. This is because in a micro-architectural study, we are interested in the maximum throughput of the processor, and we do not want to be dependent on the arrival rate of the jobs.

Random scheduler. We assume that when a job finishes, a new job is randomly selected and run on the core that has become available now. This is accurate for micro-

architectural studies that assume a basic OS scheduler. We will discuss intelligent schedulers in Section 10.

Note that our intention is to approximate a real throughput experiment as closely as possible. We assume an ideal theoretical TPEX, including first-order effects, such as the impact of longer running jobs, but ignoring other effects, such as small-grain phase behavior and transition effects. Accounting for all of these effects requires a real throughput experiment, which is just what we want to avoid. However, if one wants to measure the impact of context switches and transition effects, or if this impact is expected to be large (e.g., if jobs are short or have quickly changing phase behavior), our approach will be inaccurate and a real throughput experiment needs to be done.

3.3. Benchmark Suite Versus the Number of Job Types in a TPEX

A benchmark suite contains a number of benchmarks that show representative behavior in a certain application domain. This does not mean that all of these benchmarks are executed at the same time in a real system. The number of different job types in a real throughput experiment is usually much lower than the number of benchmarks in a benchmark suite. Therefore, we make a difference between the TPEX heterogeneity (i.e., the number of job types in a TPEX) and the number of benchmarks in a benchmark suite. TPEX heterogeneity is a setting to be chosen by the researcher, what he or she thinks is appropriate for the study.

Because the number of benchmarks is usually larger than the TPEX heterogeneity, multiple TPEXs can be constructed from a benchmark suite. If there is no knowledge about the specific jobs in the final system and all benchmarks are equally probable, all possible combinations consisting of N benchmarks (with N the TPEX heterogeneity) should be evaluated. Of course, this is practically not feasible, so only a sample of this space can be evaluated. Other studies have presented methods to select representative benchmark combinations and how to average throughput over multiple TPEXs [Velásquez et al. 2013]. Our paper specifically focuses on calculating the average throughput of one TPEX.

4. INSTANTANEOUS THROUGHPUT

We first define metrics for instantaneous throughput. As discussed in the previous section, instantaneous throughput is the throughput of a specific coschedule. Because we assume job homogeneity, the throughput of a coschedule is constant (on a large enough time scale). How to calculate average throughput using instantaneous throughput numbers is discussed in Section 5.

4.1. Unit of Work and IPC Versus Weighted IPC

Throughput is generally defined as the amount of work executed per unit of time. Time is well defined, and can be expressed in cycles (for fixed frequencies) or seconds. On the other hand, the amount of work is not unambiguously defined and depends on the context. A typical system level notion of throughput is the number of jobs or transactions finished per time unit. The unit of work is the job, which means that each job is worth the same amount of work. However, representative jobs are hard or impossible to obtain, especially in an architecture study. Therefore, we need another unit of work that is both practical and representative.

A good unit of work should be independent of the machine the job is running on and unambiguously define the size of a job expressed in that unit of work. It should allow to compare job sizes, and to define jobs with an equal amount of work. Furthermore, for a simulation-based study, it should be feasible to simulate multiple units of work in a reasonable time.

In this paper, we use two different units of work: the instruction and the instruction weighted by a reference IPC. Although this choice is inherently arbitrary, these units of work satisfy all requirements and are familiar to computer architecture researchers. Instruction counts are easy to measure, compare and even fix (by halting simulation at a certain instruction count). Obviously, a specific study can have a specific unit of work, but instructions can be used in most computer architecture studies.

Instruction as the unit of work. The most elementary unit of work is the instruction. Each instruction is worth the same amount of work and jobs have equal size if their instruction count is equal. This is a frequently used assumption in single core performance studies, and has led to the use of IPC (instructions per cycle) as a performance and throughput metric: the execution rate of a job is equal to the number of instructions (i.e., units of work) that are executed per cycle (i.e., the unit of time). IPC is proportional to job throughput (i.e., jobs finished per unit of time) if all jobs have the same size (i.e., the same number of instructions).

Although the instruction is an intuitive and straightforward unit of work, it sometimes has some unwanted consequences, especially in the context of co-running different applications. Instructions have different execution times, so applications with more short-latency instructions (e.g., ALU instructions) have a higher IPC than applications with complex instructions (e.g., floating point instructions, difficult-to-predict branches, loads causing cache misses, etc.). As a result, prioritizing high-IPC jobs results in a higher overall IPC compared to prioritizing low-IPC jobs. This conflicts with the assumption of giving equal importance to equally-sized jobs. This effect has also been noticed by previous work [Snavely and Tullsen 2000].

Weighted instruction as the unit of work. To solve the problem of having different instruction latencies, we can weight each instruction with its latency. One way to weight instructions according to their latency is to divide the IPC of a job in the multi-program experiment by the IPC of that job on a reference core. This approach is equal to the weighted IPC metrics, such as weighted speedup [Snavely and Tullsen 2000] or STP [Eyerman and Eeckhout 2008] and the harmonic mean of speedups [Luo et al. 2001] or ANTT [Eyerman and Eeckhout 2008]. Weighted IPC attributes a different amount of work to instructions of different benchmarks, proportional to their average execution time on the reference core. This is equivalent to stating that the work performed by a job in a multi-program workload is proportional to the time it takes to execute the same amount of instructions on the reference core. If all the jobs have the same execution time on the reference core, they have the same amount of work, and weighted IPC is equivalent to job throughput (i.e., the job as the unit of work).

Using weighted IPC as a throughput metric has the drawback that the results depend on the choice of the reference core. To obtain meaningful results, the reference core should be as close as possible to the core configurations in the multi-core processor under study. This is straightforward for a homogeneous multi-core, but in case of a heterogeneous multi-core, multiple reference core configurations are possible, which might have an impact on the resulting throughput number. This impact can possibly lead to different conclusions, especially if the relative performance of the benchmarks is different on the different core types. Although it is unlikely that these differences are large (e.g., a program with many cache misses will have a low IPC on both an in-order and an out-of-order core), a sanity check could be to calculate the throughput multiple times, using each of the core types as the reference.

4.2. Calculating Instantaneous Throughput

Once the unit of work (instruction or weighted instruction) and the resulting single core throughput metric (IPC or weighted IPC) is selected, the instantaneous through-

put of a coschedule executing on a multi-core processor is simply calculated as the sum of all throughput values of each core (or thread context).

Formally, we define N as the number of job types in the throughput experiment (TPEX heterogeneity), and K as the number of cores (or thread contexts). $IPS_c(s)$ is the IPS (instructions per second)[1] of the job running on core c while executing coschedule s, with s an array of K elements giving the types of the jobs running on each of the K cores. Further, $s[c]$ denotes the type of the job (an id between 1 and N) running on core c in coschedule s. IPS_b^R is the IPS of job type b on the reference core[2].

The instantaneous throughput of coschedule s using the instruction as the unit of work is then

$$i(s) = \sum_{c=1}^{K} IPS_c(s). \tag{1}$$

Instantaneous throughput with the weighted instruction as the unit of work is then calculated as follows:

$$w(s) = \sum_{c=1}^{K} \frac{IPS_c(s)}{IPS_{s[c]}^R}. \tag{2}$$

The latter equation is equal to that of weighted speedup [Snavely and Tullsen 2000] or STP [Eyerman and Eeckhout 2008]. This discussion confirms that it quantifies throughput, but only instantaneous throughput, not that of a full throughput experiment, and under the assumption that jobs are sized such that their reference core execution time is equal.

To calculate average throughput (see next section), we can use both definitions of instantaneous throughput, with the same underlying assumption. In other words, we can calculate average unweighted instruction throughput, with its assumption of equal job instruction count, or average weighted instruction throughput, assuming equal reference core execution times. The way of averaging is independent of these assumptions, meaning that in any expression in the next section for calculating average throughput based on unweighted IPS numbers, IPS can be substituted by weighted IPS (IPS divided by reference core IPS). We define $it(s)$ as the instantaneous throughput of coschedule s, which can either be $i(s)$ or $w(s)$. Similarly, $e_c(s)$ denotes the *execution rate* of the job running on core c while executing coschedule s, which can be IPS or weighted IPS.

5. AVERAGE TPEX THROUGHPUT

A throughput experiment usually consists of multiple coschedules, i.e., multiple possible combinations of job types. Each of them has an instantaneous throughput, as discussed in the previous section. We now discuss how to average the instantaneous throughput of all coschedules of the TPEX to obtain the average TPEX throughput. We first examine the implications of taking an unweighted average over all coschedules, and we find that this results in a metric where the workload is variable across experiments. We then show that fixing the workload for all experiments requires a more complex averaging, only leading to a simple formula in case of insensitive job types. Section 6 describes some techniques to obtain average fixed workload TPEX throughput for the general case.

---

[1]If all cores have a constant and equal clock frequency, IPS can be replaced by the more common IPC (instructions per cycle).

[2]Note the two different IPS notations: without superscript R and with argument (s), it denotes the IPS of a job in a coschedule; with superscript R and without (s), it is the IPS of a job when executed alone on the reference core.

5.1. Variable Workload Average Throughput

The simplest TPEX consists of only one coschedule. An example is a homogeneous workload (multiple copies of the same job type). For these TPEXs, average throughput is equal to the instantaneous throughput of the single coschedule. However, in most TPEXs, the coschedule changes over time. In our setup, all permutations (with repetition) of K (number of cores) job types out of N (workload heterogeneity) can occur, leading to N^K possible coschedules.

Ideally, we want each job type to have an approximately equal impact on the TPEX, meaning that all job types have a similar execution time in the TPEX. Having a large discrepancy in job execution times may lead to a situation where most of the time, only the long running jobs are active, effectively resulting in a TPEX with only long running jobs. Therefore, the execution time should be approximately equal for all job types, which is partly realized by assuming equal-work jobs (either equal instruction count or equal reference core execution time).

When all job types have an equal execution time, each coschedule is equally probable. A simple approach is therefore to average the instantaneous throughput over all possible coschedules:

$$AV = \frac{1}{N^K} \sum_{s=1}^{N^K} it(s) \tag{3}$$

where $it(s)$ can be either $i(s)$ or $w(s)$, leading to AVI and AVW, respectively. AV stands for Average throughput assuming a Variable workload. This naming is explained by the consequences of assuming that each coschedule is equally probable, or in other words, each coschedule is running the same amount of time, independent of individual performance of each of the jobs in the coschedule. Assuming equal execution times for all coschedules ignores relative performance differences between job types across different multicore configurations. Different multicore configurations usually have a different throughput for the same coschedule, and also different relative throughputs between coschedules (for example, coschedule x may have a larger throughput than coschedule y on one configuration, but a smaller throughput on another configuration). Fixing the time each coschedule executes means that the amount of work is different over different configurations: a fixed time and a different throughput means a different amount of work. Therefore, we call this metric a variable workload metric: the workload is not fixed across all experiments.

A variable workload metric can be misleading when comparing similar processor architectures, e.g., a baseline processor and a processor augmented with a proposed enhancement. Especially when certain jobs have better performance and others have worse performance compared to the baseline, assuming equal execution times favors the better performing jobs more. For example, if job A has an IPC of 2 and job B an IPC of 1 in one configuration, AVI assumes that job A executes twice as much instructions as job B. If for another configuration, job A now has an IPC of 1 and job B an IPC of 2, then job B executes twice as much instructions than job A, leading to a completely different workload for the two configurations and therefore uncomparable throughput values. For AVW the problem is slightly less distinct, because weighted IPC numbers are all between 0 and 1, leading to less variation between benchmarks. However, even for AVW, which is in fact the arithmetic mean of the weighted speedup of all coschedules, the workload will still depend on the configuration.

In rare cases, variable workload metrics can be used because they are easy to calculate. A straightforward case is a close to homogeneous workload, where all jobs show very similar behavior and have the same size. In that case, all jobs will execute approximately the same amount of time. Another possible case is if the processors under

comparison are completely different (for example, a current design compared to a future generation design), and it is not sure or less important that the workload is exactly the same. In this case, a variable workload metric could be used, because it enforces an equal execution time for all job types, rather than taking into account (sometimes subtle) relative performance differences between different jobs on different processors.

5.2. Fixed Workload Average Throughput

It is usually more meaningful to compare different multi-core configurations under the same workload, i.e., the relative number of instructions executed by each job type is the same for all experiments. If jobs A and B execute the same number of instructions on configuration 1, they should also execute the same number of instructions on configuration 2, even if their relative performance differs. The Average throughput assuming a Fixed workload (AF) is

$$AF = \frac{\text{total TPEX work}}{\text{total TPEX time}} \tag{4}$$

Here, the work executed is the same for all multicore configurations, and only the time to execute the work varies. Throughout this study, we consider AF throughput metrics with the following common assumptions:

— All job types are equally likely.
— The job size is the same for all job types.

The job size is defined from the chosen unit of work. We consider two AF metrics in this study: AFI and AFW. AFI assumes that all the jobs execute a fixed number of instructions, AFW assumes that all the jobs execute a fixed number of weighted instructions. As indicated before, a specific study can have different assumptions, leading to different metrics.

Note that because of the assumption of a fixed workload, we enforce fairness between job types, i.e., each job type executes a fixed amount of work. In this paper, we assume an equal amount of work for each job type, but it is straightforward to assign different weights to different job types if needed. Furthermore, a fixed workload metric makes sure that no job type is favored, as opposed to the variable workload metrics, that favor faster running job types. Therefore, it does not matter whether throughput is expressed in IPC or weighted IPC, the only difference between AFI and AFW is the size of the jobs. To the best of our knowledge, fixed workload average throughput metrics have never been proposed or used before.

In theory, the AF throughput can be computed as a weighted arithmetic mean of instantaneous throughput over all coschedules, the weights being the probability for each coschedule to occur in time. However, in general, there is no simple formula for computing these probabilities, because the execution rate of a single job in a specific coschedule may impact the probabilities of all coschedules. For example, speeding up job A in coschedule 'AB' can result in a higher probability for coschedule 'BB'.

However, there is one situation where it is feasible to find a closed expression, namely in case the performance of each of the jobs is independent of the types of the co-running jobs. In other words, in every coschedule, a specific job type always has the same performance (which might be different from its performance on the reference core due to interference). In case of a heterogeneous multicore, the performance of a job type on the different core types is different, but for each core type, the performance of a job type is constant. Formally,

$$\forall s, c : IPS_c(s) = IPS_{c,b} \text{ with } b = s[c] \tag{5}$$

i.e., the IPS for job type b is equal for all coschedules where b is run on core c. We call these jobs *insensitive*.

For insensitive jobs, we can calculate the throughput per core, because the performance of the jobs does not depend on what is running on the other cores. The total throughput of running all N job types on a core equals (with T_b the execution time of job type b, and $e_{c,b} \propto \frac{1}{T_b}$ its execution rate)

$$\frac{N}{\sum_{b=1}^{N} T_b} \propto \frac{N}{\sum_{b=1}^{N} \frac{1}{e_{c,b}}} \tag{6}$$

The throughput of a core is thus the harmonic mean of the execution rate of all jobs on that core. The execution rate $e_{c,b}$ can be either the $IPS_{c,b}$ or $\frac{IPS_{c,b}}{IPS_b^R}$, depending on the choice of the unit of work (instruction or weighted instruction). The average throughput of the processor is then the sum of the throughputs of all cores.

For the instruction as unit of work, we obtain

$$AFIi = \sum_{c=1}^{K} \frac{N}{\sum_{b=1}^{N} \frac{1}{IPS_{c,b}}} \tag{7}$$

(where the appended "i" reminds that the formula is exact for insensitive jobs only). For the weighted instruction as the unit of work, we get

$$AFWi = \sum_{c=1}^{K} \frac{N}{\sum_{b=1}^{N} \frac{IPS_b^R}{IPS_{c,b}}} \tag{8}$$

When all cores are identical, equation 7 is similar to the harmonic mean of IPCs (which was proposed as a multi-program throughput metric in [Michaud 2013]) and equation 8 is similar to the harmonic mean of speedups (proposed in [Luo et al. 2001], and inversely proportional to ANTT [Eyerman and Eeckhout 2008]). Here, we show that they both quantify throughput when jobs are insensitive and assuming jobs have an equal instruction count or equal reference core execution time, respectively. The harmonic mean of speedups was proposed as a fairer metric than weighted speedup, by taking the harmonic mean instead of the arithmetic mean, which gives more weight to low speedups. We deduce that its fairness lies in the (implicit) assumption of a fixed workload: a slower job will execute longer, and therefore has a greater impact on total throughput than a fast job that finishes quicker.

However, jobs are in general sensitive to which jobs are co-running. These formulas do not hold for sensitive jobs, for which there is no single $IPS_{c,b}$. Only in case all $IPS_c(s)$ where $s[c] = b$ are all close to each other, Equations 7 or 8 produce an accurate throughput number. In that case, $IPS_{c,b}$ can be calculated as the harmonic mean over all $IPS_c(s)$ where $s[c] = b$. In Section 8 we will show that these formulas are not always a good approximation for average throughput.

So the question remains how to accurately calculate average throughput in case of a fixed workload and sensitive jobs. So far, we have found closed formulas for some specific cases only, which are too restricted to be used in realistic experiments. In the next section, we elaborate two techniques to calculate average throughput for the general case: Monte Carlo simulations and Markov Chains. These do not provide closed formulas, but use iterative methods to calculate average throughput. Monte Carlo simulations are easy to understand and implement, but can take some time to converge. Markov Chains are solved quicker, and can produce closed expressions for simple cases. The theory behind it is, however, less intuitive.

We conclude this section with an overview of all metrics (see Figure 2). The figure shows the execution profile of the TPEX assumed by the metrics, using the example

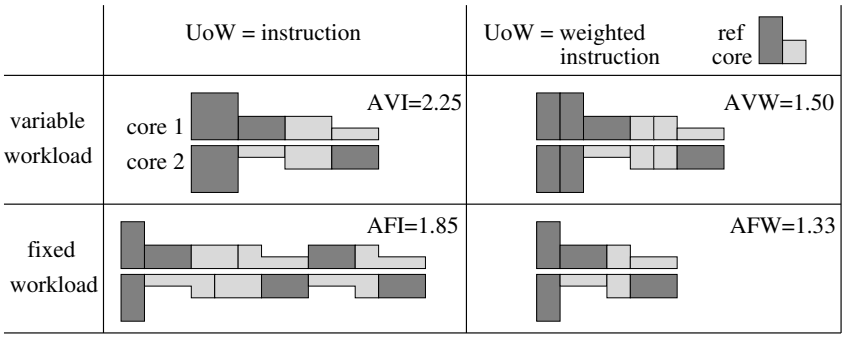

Fig. 2. Graphical overview of the metrics.

from Table I. For the variable workloads, the coschedule execution time is equal for all coschedules but the amount of work executed by each job type differs (e.g., job A executes more instructions than job B for AVI). For the fixed workload metrics, some coschedules are executed longer than others, but the amount of work is the same for all job types.

6. TECHNIQUES TO DERIVE FIXED WORKLOAD METRICS

Average fixed workload (AF) throughput metrics, i.e., metrics that assume that the same instructions are executed for all experiments, are difficult to calculate because the probability of a coschedule to occur may depend on the execution rates of all jobs in all coschedules. In this section, we present two methods to calculate these metrics: Monte Carlo simulations and Markov Chains. Because these methods are not as easy to use as closed formulas, we implemented both methods in a tool. The tool takes the IPS (IPC) numbers from the simulations, and produces the average throughput using a method of choice (Monte Carlo simulations or Markov Chain analysis). We encourage using this tool to evaluate the throughput of a multi-program workload, as it provides the most realistic throughput numbers for the most general setup. Section 7.2 provides more information about the tool.

6.1. Monte Carlo Simulations

Instead of trying to find a closed expression, the AF throughput can be obtained by simulating the corresponding TPEX with a method similar to the co-phase matrix method introduced by Van Biesbrouck et al. [Van Biesbrouck et al. 2004; Van Biesbrouck et al. 2006]. We start with a random coschedule where each job is set to the same size. Because we assume that the IPS values of all the coschedules are known, we can easily compute the time until the next job termination. The terminated job is then replaced with a new job whose type is randomly selected among the N possible job types. Then we calculate the time until the termination of the next job, and so on. By simulating a sufficiently large number of jobs, we can approach the AF throughput value asymptotically.

Monte Carlo simulations are different from real TPEX simulations in that we do not simulate each job in detail: we assume its IPS equals the IPS measured in the isolated simulations. This means that the final throughput number can be calculated fairly quickly, as soon as the IPS values of all coschedules are known. Nevertheless, it can take some time until the final number converges, especially when there are many job types. After convergence, a Monte Carlo simulation provides the correct AF throughput, under the assumptions of this metric. This means that this method can be used to validate other (faster) techniques.

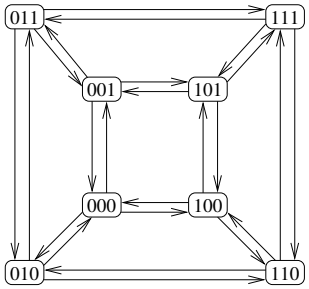

Fig. 3. CTMC for the AF metrics, for K=3 cores and N=2 job types. There are 8 states, one per coschedule. A transition from a coschedule s to a coschedule s' happens when a job finishes on a core c. The transition rate from s to s' is equal to the execution rate (e.g., IPS) on core c in coschedule s, divided by the number N of job types.

6.2. Continous-time Markov Chain

The AF throughput can also be obtained by modeling the corresponding TPEX as a continuous-time Markov chain (CTMC) [Pinsky and Karlin 2010]. Unlike the AF assumption of constant job size, the CTMC method requires that job sizes are exponentially distributed, with the same average for all job types. We verified with Monte Carlo simulations that in practice, exponentially distributed job sizes and constant job sizes give very close throughput values. The main advantage of the CTMC method is that some insight can be obtained from it.

Each of the N^K possible coschedules corresponds to a state of the CTMC. In the remaining, *state* is synonymous with *coschedule*. Figure 3 shows the CTMC for $K = 3$ cores and $N = 2$ job types. A transition from a coschedule s to a coschedule s' happens when a job finishes on a core c. This job is replaced by a new job, chosen randomly and with uniform probability from the N job types (or following a given distribution of job weights). The transition rate from s to s' is proportional to the execution rate (IPS, IPC, weighted IPC, etc.) on core c in coschedule s, divided by the number N of job types. This can be explained as follows. We assume an exponentially distributed (with equal average) job size for all cores, given the unit of work. The execution rate has the same unit of work, which means that average job execution time is inversely proportional to the execution rate. Due to the memorylessness property of the exponential distribution, the remaining time of the job on core c does not depend on the previous states, so the time to finish is proportional to the reciprocal of the execution rate. Therefore, the transition rate from this state to another state where c has a new job is equal to the execution rate of c. Because each job type is equally likely, the transition rate to a specific state with another job on core c is equal to the execution rate divided by N.

Each coschedule has a certain steady-state probability to occur in time, which is independent of the initial state. Steady-state probabilities can be obtained by solving a system of N^K equations in N^K unknowns. These equations express the fact that the sum of all state probabilities is one, and that we enter any state as often as we leave it. The AF throughput is then obtained as a weighted arithmetic mean of all N^K instantaneous throughputs, the weights being the steady-state probabilities.

When considering a small number of cores and job types, closed-form formulas may be obtained. For instance, for a symmetric dual-core, two job types, and denoting IPC_{ij} the IPC of job type i when co-running with job type j, the AFI throughput is

$$\text{AFI} = \frac{4 \times (IPC_{12} + IPC_{21})}{2 + \frac{IPC_{21}}{IPC_{11}} + \frac{IPC_{12}}{IPC_{22}}}$$

The AFW throughput is obtained by replacing IPCs with weighted IPCs. The same formula has been deduced by Michaud [Michaud 2012] using a totally different approach, which assumed fixed job sizes. In this case, assuming exponentially distributed job sizes results in exactly the same formula as for fixed job sizes.

When the number of cores or job types is not small, the closed formula becomes very complex, and it is more practical to solve the system of equations numerically. A detailed description of the method is given in appendix A.

6.3. Properties of the AF Metrics

Because there is no simple formula for computing the AF throughput, we need some intuition to understand the results we will obtain from them. We show in appendix A that the AF throughput can be computed as a weighted harmonic mean:

$$\text{AF} = \frac{1}{\sum_{s\in[1,N]^K} \frac{w_s}{it(s)}} \tag{9}$$

where the positive weights w_s depend solely on the *instantaneous throughput fractions*, which is the execution rate $e_c(s)$ of core c in coschedule s, divided by the instantaneous throughput $it(s) = \sum_{c=1}^{K} e_c(s)$:

$$\text{instantaneous throughput fraction} = \frac{e_c(s)}{it(s)}$$

From this insight, we can deduce some interesting properties.

Uniform acceleration. Let us take a particular coschedule s, and let us apply a *uniform acceleration* on it: that is we accelerate each of the K execution rates by the same factor $\sigma > 1$. In this case, the instantaneous throughput fractions and the weights w_s are left unchanged, but the instantaneous throughput $it(s)$ has been multiplied by σ. Hence from (9), the AF throughput increases. Because it is a harmonic mean, uniformly accelerating a coschedule is likely to increase the AF throughput more if the accelerated coschedule has a low instantaneous throughput. In other words, *a method that guarantees increasing the AF throughput is to increase uniformly the instantaneous throughput of some coschedules, where "uniformly" means that all the jobs in a coschedule are accelerated by roughly the same factor. For greater impact, we should first try to accelerate coschedules with a low instantaneous throughput.*

Note that non-uniformly increasing instantaneous throughput of a coschedule (e.g., by accelerating one job and slowing down another) has an impact on the weights, which can possibly result in a lower average throughput.

Equal execution rates. We also show in appendix A that, when the instantaneous throughput fractions are all equal to $1/K$, all the weights w_s are equal to N^{-K}, so *in the particular case when, for each coschedule, the execution rates in that coschedule are all equal, the AF throughput is the* **unweighted** *harmonic mean of the instantaneous throughputs over all coschedules. Consequently, the AFIi and AFWi approximations are exact in this particular case.*

This can occur when all job types have very similar behavior and therefore have approximately equal execution rates. Another example is a processor that enforces fairness by equalizing per-thread slowdown: by construction, each job has the same weighted IPC, so AFWi equals AFW.

7. PRACTICAL CONSIDERATIONS

Now that we have explored the theoretical background of variable and fixed workload throughput metrics, in this section, we provide practical advice on how to use these

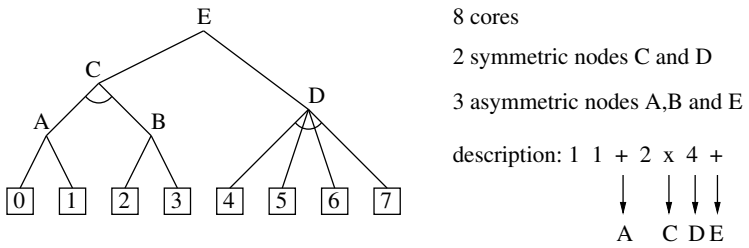

Fig. 4. The *TPCalc* software exploits multicore symmetries. This is an example with 8 cores. Nodes A and B are identical, cores 4,5,6,7 are identical. The description language uses reverse polish notation.

metrics in a multi-program performance study. In the next sections, we discuss how to simulate the coschedules, how to exploit symmetries and how to use our tool, and the possibilities to use a subsample of the coschedule space.

7.1. Simulating the Coschedules

The first step in evaluating throughput is simulating the individual coschedules. In the elaboration of the metrics, we assumed job homogeneity. This means that jobs or benchmarks have a constant behavior, which can be achieved through selecting homogeneous phases in existing benchmarks, and treat them as distinct job types. Furthermore, we also assume a constant behavior while co-executing multiple programs. This means that a simulation of a coschedule should provide stable average IPS numbers. In reality, there are small phases in every program, so in general, we should avoid simulating only a part of a benchmark or a part of a representative phase. An important factor here is how and when to stop simulating. The simulation can stop when the first job ends, but then the other jobs are not fully simulated. Continuing simulating the other jobs does not record the interference with the job(s) that already finished. Several methods have been proposed to solve this problem, from restarting jobs until the slowest job finishes, to keep on restarting jobs until the average IPC converges [Vera et al. 2007]. The simulation methodology is conceptually orthogonal to the calculation of the metrics, but in general, the IPS numbers will be more stable when each job is fully executed at least one time.

7.2. Symmetries and the *TPCalc* Tool

If all the cores in a multicore are different, we need all the N^K coschedules to compute the AF throughput. However, in general, some cores are identical. By exploiting multicore symmetries, we can significantly reduce the number of coschedules to simulate.

We have written a software tool called *TPCalc* (throughput calculator) that allows a user to describe the symmetries in a multicore, using reverse polish notation. Figure 4 gives an example of a multicore configuration, and the corresponding description.

The *TPCalc* tool uses command line parameters to control its functionality. Without any parameters, it outputs the configuration and the number of base coschedules. For example, the following command is for 2 job types and a fully symmetric 3-core:

```
tpcalc 2 "3"
```

The output of this command is

```
3 logical cores
4 base coschedules
These cores are identical: 0 1 2
```

The user can also obtain the list of base coschedules to simulate. This command

```
tpcalc 2 "3" -list
```

gives the 4 base coschedules:

```
0 :  0  0  0
1 :  1  0  0
2 :  1  1  0
3 :  1  1  1
```

The zeros and ones correspond to the first and second benchmark respectively. Then the user chooses 2 benchmarks, simulates the 4 base coschedules, and provides to *TPCalc* the 12 execution rates (IPS, IPC, weighted IPC, etc.), in the same order. For instance, the following command gives the AVI and AFI throughputs, assuming the 12 execution rates are written in a file RATESFILE:

```
tpcalc 2 "3" -metric AVI -metric AFI < RATESFILE
```

All metrics described in this paper can be computed with *TPCalc*. The *TPCalc* software is available for download at http://www.irisa.fr/alf/downloads/michaud/tpcalc.html.

7.3. What if there is a Large Number of Base Coschedules ?

In general, the number of different job types running simultaneously on a server is limited. So a common throughput experiment has a relatively small workload heterogeneity, e.g., $N = 2$. When K is relatively small, the number of base coschedules is reasonable. For instance, $N = 2$ job types and four 4-way SMT cores ($K = 16$ logical cores) leads to 70 base coschedules. The 70 microarchitecture simulations, one for each coschedule, can be done in parallel. However, the number of base coschedules increases quickly with the number of cores. For example, $N = 2$ job types and eight 4-way SMT cores ($K = 32$ logical cores) leads to 495 base coschedules. These coschedules can be evaluated in parallel, which makes this method still faster than a real TPEX, where the coschedules are evaluated sequentially.

For a large experiment, the time or compute power required to run all simulations can become (too) big. To reduce the simulation time, one can simulate a sample of all coschedules. However, not all metrics lend themselve well to sampling. We can use random sampling with the AVI and AVW metrics because each coschedule has the same weight. The more samples are simulated, the closer the throughput is to the real throughput [Velásquez et al. 2013].

The AF metrics, on the other hand, cannot be sampled, because the weight of each coschedule depends on the execution rates of all coschedules. With the CTMC method, all the base coschedules are necessary for computing the AF throughput, because the state probabilities are obtained by solving a system of equations. If a single coschedule is missing, we cannot solve the system.

In last resort, it is possible to use the AFIi and AFWi approximations. Because in AFIi and AFWi, all jobs also have the same weight, they lend themselves to sampling, and they can be estimated from a sample of coschedules. If the jobs in the workload are quasi insensitive, or if the instantaneous throughput fractions are all approximately equal (cf. Section 6.3), the AFIi and AFWi approximations are accurate. Otherwise, there is no guarantee that AFIi and AFWi give meaningful throughput values, as we will see in the next section.

8. COMPARING THROUGHPUT METRICS

By definition, different throughput metrics may give different conclusions and may lead to different design decisions. Solutions that a computer architect may propose to increase multi-program throughput generally come from some initial intuitions. We believe that, most of the time, these are intuitions about instantaneous throughput.

Table II. Selected SPEC CPU 2006 benchmarks

Benchmark	Input	Benchmark	Input
bzip2	input.program	libquantum	ref
calculix	ref	mcf	ref
gcc	cp-decl.i	perlbench	diffmail.pl
gcc	g23.i	sjeng	ref
h264ref	foreman_ref_encoder_main	tonto	ref
hmmer	nph3.hmm swiss41	xalancbmk	ref

For instance, if we think of a way to increase the IPC of a job without decreasing that of the co-running jobs, there is a good chance that we increase instantaneous throughput, whatever the chosen unit of work. And in many cases, an increase of instantaneous throughput yields an increase of average throughput, whatever the throughput metric. We believe that, in many situations, different throughput metrics give similar qualitative conclusions and lead to the same design decisions. Nevertheless, there may be some situations where this is not the case.

If the assumptions behind one particular throughput metric correspond precisely to the situation being studied, it may be sufficient to use that single metric. However, we often lack precise information about real workloads, and it is common practice in computer architecture studies concerned with multiprogram workloads to use several different metrics at once. If several different metrics yield the same conclusion, this gives more confidence in the conclusion. If different metrics give different conclusions, this may be a warning about the uncertainty of the conclusion.

In this section, we present an example study on different simultaneous multithreading (SMT) fetch and resource sharing policies. The purpose of this example is not to discover new "truths" about microarchitecture, but to show that different throughput metrics may give different conclusions.

We consider 4 threads running concurrently on a 4-wide out-of-order SMT core. We consider two fetch policies:

— Round robin (RR): the core fetches from threads that are able to fetch (i.e., that have no I-cache miss or a full private resource) by turns.
— ICOUNT: the core fetches from the thread that is able to fetch and that has the least in-flight instructions [Tullsen et al. 1996].

and two reorder buffer partitionings:

— Static: each thread has one fourth of the reorder buffer [Raasch and Reinhardt 2003].
— Dynamic: the reorder buffer is shared between all threads in an unsupervised way, i.e., the number of instructions per thread depends on the fetch and execution rate of that thread.

We simulated all possible 4-program workloads that can be constructed using 12 selected SPEC CPU 2006 benchmarks (see Table II; we selected these benchmarks out of 53 benchmark-input pairs based on their SMT behavior, they approximately uniformly cover the space of low- to high-interference benchmarks), using the IWC core model in Sniper [Carlson et al. 2011] (simulating the reorder buffer, functional units, branch predictor and caches in detail). For each of the 1365 workloads, we did four simulations for all combinations of the fetch policy and resource partitioning: RR-Static, RR-Dynamic, ICOUNT-Static and ICOUNT-Dynamic.

Figure 5 shows the fraction of workloads for which a specific configuration is optimal, using the metrics on the X-axis. It is clear that AVI differs most from the rest: for more than 60% of the workloads, ICOUNT-Static is optimal, while the other metrics point to ICOUNT-Dynamic as being optimal for the majority of the workloads. The reason for this is that when a thread encounters a last-level cache (LLC) miss (to main memory),

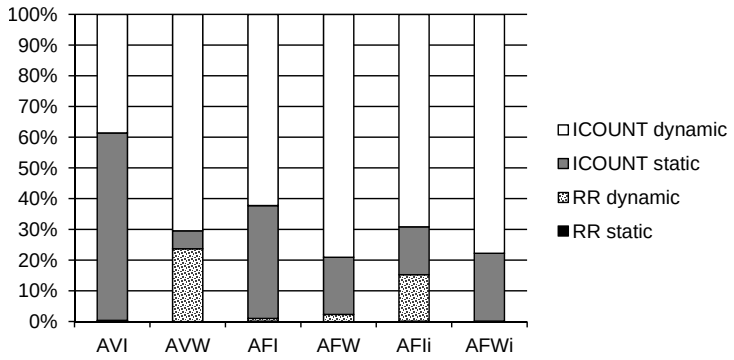

Fig. 5. Fraction of workloads that have a specific configuration as the optimum, using different throughput metrics.

the miss will prevent committing instructions from that thread. ICOUNT ensures that normally, no instructions will be fetched as soon as the thread occupies one quarter of the ROB, even for dynamic ROB partitioning. However, in case the other threads have front-end misses and cannot fetch instructions, instructions will be fetched from the thread that has a miss. These instructions continue to occupy ROB entries, leading to fewer entries for the other threads (which is not the case for ICOUNT-Static). This observation is also made by Tullsen et al. [Tullsen and Brown 2001]. High-IPC jobs (that do not have many misses) can therefore have lower performance for ICOUNT-Dynamic, while jobs with many misses can perform better, because they occupy more ROB entries and can therefore exploit more MLP [Glew 1998]. The net result is that the sum of IPCs is lower (AVI), but the sum of weighted IPCs (AVW) can be higher. For the fixed workload metrics, we also showed that increasing the performance of low performance jobs has more effect than for high performance jobs, explaining their preference for ICOUNT-Dynamic.

AVW selects RR-Dynamic to be optimal for more than 20% of the workloads, while the others (except for AFIi) have only a marginal fraction that prefers RR-Dynamic. RR-Dynamic has the same problem as ICOUNT-Dynamic, namely that the ROB can be largely filled with instructions from a thread that has a LLC miss. It is even more pronounced, as the fetch policy keeps on fetching from that thread, regardless of what is already in the ROB. However, a few benchmarks that have a high LLC miss rate also have a high memory-level parallelism (MLP) if they have more ROB entries, leading to a larger weighted IPC. So the sum of weighted IPCs (AVW) can be larger for RR-Dynamic than for other policies for certain coschedules. However, this leads to a dramatic performance decrease for the other jobs, increasing the probability of the coschedules that contain these other jobs for fixed workload metrics. This explains why the fixed workload metrics (e.g., AFW) prefer RR-Dynamic less.

In conclusion, none of the metrics provide exactly the same result. This experiment also shows that AFIi is a bad approximation for AFI. AFW and AFWi are close for this particular experiment, but as discussed previously, this observation cannot be generalized.

9. ENERGY AND POWER

Energy consumption is an important concern in modern processors. It is important to be able to quantify the average power of a multi-core processor running multi-program workloads.

Similarly to throughput, we distinguish per-coschedule “instantaneous” power and average power. Per-coschedule power is typically what a microarchitecture simulator

featuring a power model can provide. It is unambiguously defined. Average power is characterized by a TPEX: to each throughput metric, we can associate an average power metric. Denoting $p(s)$ the per-coschedule power (i.e., the total power consumption of the processor running all jobs in the coschedule), the average power P is:

$$P = \sum_{s \in [1,N]^K} \pi_s \times p(s) \tag{10}$$

where π_s is the probability for the occurrence in time of coschedule s. The probabilities π_s depend on the assumptions of the TPEX. If we assume a variable workload (AVI/AVW metrics), the average power must be computed as the unweighted arithmetic mean of per-coschedule instantaneous powers ($\pi_s = N^{-K}$). However, if we assume a fixed workload (AFI/AFW metrics), the average power is a weighted arithmetic mean of per-coschedule instantaneous powers, and the weights π_s are obtained by solving the system of equations (12) and (16) (see appendix A).

10. LIMITATIONS AND FUTURE WORK

10.1. Multiple Multi-Threaded Programs

So far we have considered workloads consisting of multiple independent single-threaded programs. A multi-program workload can also consist of multiple multi-threaded programs (or a mix of single- and multi-threaded programs). The main difficulties with multi-threaded programs are choosing a meaningful unit of work, and measuring instantaneous throughput.

For multi-threaded programs that contain spin-wait or spin-lock loops, the instruction is not a safe unit of work [Alameldeen and Wood 2006]. The instruction can be used if we can identify spin loops and ignore their instructions when reporting IPCs, considering only useful instructions.

The weighted instruction might be considered a more convenient unit of work for multi-threaded programs, as the weighted instruction is akin to speedup. Indeed, speedup is defined unambiguously for all multi-threaded programs, provided we run programs to completion. However, running programs to completion is often infeasible when doing microarchitecture simulation, because of the slow simulation speed. In practice, we want to measure instantaneous throughput on an execution sample. Sampled simulation for multi-threaded programs is an active area of research and is out of the scope of this study (see the solution recently proposed by Carlson et al. [Carlson et al. 2014]). Once a unit of work is defined, this defines an instantaneous throughput metric. For instance, Creech et al. use weighted IPC as instantaneous throughput metric for multiple multi-threaded programs running concurrently on a multicore [Creech et al. 2013].

Meaningful average throughput metrics for multi-threaded jobs cannot be defined exactly like for single-threaded jobs. For single-threaded jobs, it is sufficient to assume that the job arrival rate is high enough such that cores are never idle, because we seek to quantify the maximum throughput. For multi-threaded jobs, the number of threads per job may vary depending on the number of jobs competing for execution [Creech et al. 2013]. Defining meaningful average throughput metric for multi-threaded jobs will probably require to model job arrivals.

10.2. Fairness

To maximize throughput, it might be necessary to sometimes slow down some of the jobs. A reasonable system must consider not only throughput but also *fairness*: the extent to which an individual job can be slowed down while competing with concurrent jobs for shared resources should be limited. Fairness is essentially an “instantaneous”

Table III. Metrics overview and recommendations

Study	*Metric*	*Comment*	*Drawback*
Designing schedulers Very short jobs Lots of context switches	Real throughput experiment	Faithfully models transition effects and OS overhead	Difficult to set up Many experiments needed
Microarchitecture studies:			
General case	AFI AFW	Jobs have equal instruction count Jobs have equal reference core execution time	No sampling possible
Insensitive jobs Similar jobs	AFIi, AFWi	E.g., low degree of HW sharing	Inaccurate for sensitive jobs
Homogeneous workloads Very distinct processors	AVI, AVW	Future processors Future workloads	Variable workloads across experiments!

notion, it applies to jobs that are currently running. The two instantaneous metrics we have considered in this study, IPS and weighted IPS, do not take into account fairness. Therefore, instantaneous and average throughput metrics must be complemented with a fairness metric. Several different fairness metrics have been proposed [Eyerman and Eeckhout 2008; Vandierendonck and Seznec 2011].

10.3. Intelligent Schedulers

All metrics in this paper assume a random scheduler. All jobs can run on all core types, independent of the co-running jobs. As a result, all coschedules have *some* probability to occur. An intelligent scheduler that tries to maximize throughput by scheduling jobs on core types based on their properties will have an impact on these probabilities and result in another average throughput number. For example, if the scheduler optimizes instantaneous throughput, coschedules with a low instantaneous throughput will have a lower or zero probability to occur. Note that the instantaneous throughput metrics are still valid for intelligent schedulers, and the same taxonomy (variable and fixed workload metrics) can be used to develop metrics assuming an intelligent scheduler.

10.4. Comparing Processors with Different ISAs

In this paper, we assumed that all processors in a study have the same ISA, such that they can all execute the same binaries and the instruction count per benchmark is fixed. Comparing processors with a different ISA (e.g., comparing the throughput of an Intel processor to that of an ARM processor) is more difficult, because each benchmark now needs to be compiled multiple times, once for each ISA. This means that the throughput numbers are also impacted by the optimization capabilities of the compilers. Furthermore, different versions of the same benchmark have different instruction counts, so comparing raw IPS values directly makes no sense. To solve this, the ratio between the instruction count on one ISA and the instruction count on another ISA can be used to weight (or normalize) the IPS.

11. CONCLUSIONS

There is no single correct throughput metric for multi-program workloads. Throughput depends on assumptions on job size, job distribution, scheduling, etc. A set of assumptions defines a throughput experiment, and the corresponding metric is the throughput of that experiment. Different assumptions lead to different metrics, which can have an impact on the conclusions of a study. Therefore, we should use metrics whose assumptions are plausible for a particular study.

We deduce multiple throughput metrics based on general and fair assumptions. In particular, we make a distinction between variable workload metrics and fixed work-

load metrics. We identify the assumptions behind commonly used metrics, such as the IPC throughput, weighted speedup and the harmonic mean of speedups. We also propose new fixed workload throughput metrics (AFI and AFW), that cannot always be calculated using a closed formula. We explain how to use these metrics, and provide insight into how metrics are related and how to optimize for a certain metric using experimental data. Table III summarizes the metrics we discussed and shows in which context they can be used and what the drawbacks are.

Our study can be used to select a meaningful metric for a multi-program study and serve as a starting point to define a new metric for a specific setup. Understanding the assumptions behind a metric and its limitations leads to a better understanding of the actual results of an experiment.

APPENDIX

A. MODELING AF METRICS AS A CTMC

This is a more detailed description of the CTMC method introduced in Section 6.2. We assume that the jobs sizes follow an exponential distribution, and that the average job size for each job type is equal to 1 (in the chosen unit of work). We define $e_c(s)$ as the execution rate for the job on core c in coschedule s. The execution rate is expressed in units of work per second. That unit of work is the instruction (AFI) or the weighted instruction (AFW). The instantaneous throughput $it(s)$ of coschedule s is

$$it(s) = \sum_{c=1}^{K} e_c(s) \tag{11}$$

Each coschedule corresponds to a state in the CTMC (in this appendix, *state* and *coschedule* are synonymous). Let π_s denote the steady-state probability for the occurrence in time of coschedule s. We have

$$\sum_{s \in [1,N]^K} \pi_s = 1 \tag{12}$$

The AF throughput is

$$\mathbf{AF} = \sum_{s \in [1,N]^K} \pi_s \times it(s) \tag{13}$$

We define $D(s, s')$ as the Hamming distance between two coschedules s and s'. That is, $D(s, s')$ is the number of cores c such that $s[c] \neq s'[c]$. The *neighbors* of a state s are the s' such that $D(s, s') = 1$. Each coschedule has $K \times (N - 1)$ neighbors. When a job finishes, a transition occurs from the current coschedule to one of its neighbors. $\mathcal{N}_c(s)$ is the set of neighbors of s that differ from s on core c. We have $|\mathcal{N}_c(s)| = N - 1$.

The transition rate $r(s, s')$ from a coschedule s to $s' \in \mathcal{N}_c(s)$ is

$$r(s, s') = \frac{e_c(s)}{N} \tag{14}$$

That is, if at time t we are in state s, the probability to be in state s' at time $t + dt$ is $\frac{1}{N} e_c(s) dt$. This stems from from the exponentially distributed job size with mean 1, and from all job types being equally likely[3]. Probabilities π_s follow global balance equations

[3]For instance, if the average job size is J instructions, with J large, exponentially distributed job sizes means that the probability that any instruction terminates a job is $1/J$. In a cycle, we execute IPC instructions of a job. The probability that the job does *not* terminate in that cycle is $(1 - \frac{1}{J})^{IPC} \approx 1 - \frac{IPC}{J}$. Hence the probability that the job terminates is $\frac{IPC}{J}$. Because the next job is one of the N job types, the transition rate to a specific state is proportional to $1/N$.

which express the fact that we leave a state as frequently as we enter it:

$$\forall s, \sum_{s'|D(s,s')=1} \pi_s \times r(s,s') = \sum_{s'|D(s,s')=1} \pi_{s'} \times r(s',s) \tag{15}$$

Introducing (14) in equation (15) and after a simplification, the global balance equations become:

$$\forall s,\ (N-1) \times it(s) \times \pi_s = \sum_{c=1}^{K} \sum_{s' \in \mathcal{N}_c(s)} e_c(s') \times \pi_{s'} \tag{16}$$

The probabilities π_s can be obtained by solving the system of equations (12) and (16). Nevertheless, let us rewrite these equations using the following positive quantities:

$$f_c(s) \triangleq \frac{e_c(s)}{it(s)} \tag{17}$$

$$x_s \triangleq \pi_s \times it(s) \tag{18}$$

Quantity $f_c(s)$ is the *instantaneous throughput fraction* of core c in coschedule s (notice that $\sum_{c=1}^{K} f_c(s) = 1$). Equations (13), (12) and (16) become:

$$\mathbf{AF} = \sum_{s \in [1,N]^K} x_s \tag{19}$$

$$\sum_{s \in [1,N]^K} \frac{x_s}{it(s)} = 1 \tag{20}$$

$$\forall s,\ (N-1) \times x_s = \sum_{c=1}^{K} \sum_{s' \in \mathcal{N}_c(s)} f_c(s') \times x_{s'} \tag{21}$$

The x_s are obtained by solving the system of equations (20) and (21), that is, $N^K + 1$ equations in N^K unknowns. Actually, the N^K equations (21) are not independent: the $N^K \times N^K$ matrix $\mathbf{B}$ corresponding to the homogeneous system (21) has rank $N^K - 1$. Vector $\mathbf{x} = (x_s)$ is in $\ker(\mathbf{B})$, which has dimension 1. Let $\mathbf{w} = (w_s)$ be the solution of (20) and (21) when $it(s) = 1$ for all s (that is, w_s is the coschedule probability π'_s corresponding to execution rates $e'_c(s) = f_c(s)$). Note that w_s is positive and

$$\sum_{s \in [1,N]^K} w_s = 1$$

Vector $\mathbf{w}$ is a basis of $\ker(\mathbf{B})$, hence there exists a scalar λ such that

$$\mathbf{x} = \lambda \mathbf{w} \tag{22}$$

The average thoughput is

$$\mathbf{AF} = \sum_{s \in [1,N]^K} \lambda w_s = \lambda \tag{23}$$

We can solve (20) for λ and we obtain

$$\mathbf{AF} = \frac{1}{\sum_{s \in [1,N]^K} \frac{w_s}{it(s)}} \tag{24}$$

That is, AF can be expressed as a weighted harmonic mean of instantaneous throughput over all coschedules. The important fact is that *the weights depend solely on the instantaneous throughput fractions*.

In the particular case when the execution rates in any coschedule are all equal ($f_c(s) = \frac{1}{K}$ for all c and s), the weights w_s are all equal to N^{-K}, and AF is the *unweighted* harmonic mean of the instantaneous throughputs over all coschedules.

REFERENCES

A. R. Alameldeen and D. A. Wood. 2006. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro* 26, 4 (July 2006), 8–17.

T. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.

T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. 2014. BarrierPoint: sampled simulation of multi-threaded applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

A. Carlton. 1995. CINT92 and CFP92 Homogeneous Capacity Method Offers Fair Measure of Processing Capacity. http://www.spec.org/cpu92/specrate.txt. (1995).

T. Creech, A. Kotha, and R. Barua. 2013. Efficient multiprogramming for multicores with SCAF. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 334–345.

S. Eyerman and L. Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28, 3 (May 2008), 42–53.

S. Eyerman and L. Eeckhout. 2010. Probabilistic job symbiosis modeling for SMT processor scheduling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 91–102.

S. Eyerman and L. Eeckhout. 2013. Restating the case for weighted-IPC metrics to evaluate multiprogram workload performance. *IEEE Computer Architecture Letters* (April 2013).

A. Glew. 1998. MLP yes! ILP no!. In *ASPLOS Wild and Crazy Idea Session*. 26–34.

A. Hilton, N. Eswaran, and A. Roth. 2009. FIESTA: a sample-balanced multi-program workload methodology. In *Workshop on Modeling, Benchmarking and Simulation (MoBS)*.

L. K. John. 2006. Aggregating Performance Metrics over a Benchmark Suite. In *Performance Evaluation and Benchmarking*, L. K. John and L. Eeckhout (Eds.). CRC Press, 47–58.

J. L. Kihm, T. Moseley, and D. A. Connors. 2005. A mathematical model for accurately balancing co-phase effects in simulated multithreaded programs. In *Workshop on Modeling, Benchmarking and Simulation (MoBS)*.

R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *International Symposium on Computer Architecture (ISCA)*. 81–92.

K. Luo, J. Gummaraju, and M. Franklin. 2001. Balancing thoughput and fairness in SMT processors. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 164–171.

P. Michaud. 2012. *Constant-work Multiprogram Throughput Metrics for Microarchitecture Studies*. Technical Report RR-8150. INRIA.

P. Michaud. 2013. Demystifying multicore throughput metrics. *Computer Architecture Letters* 12, 2 (July 2013), 63–66.

H. H. Najaf-abadi and E. Rotenberg. 2009. The importance of accurate task arrival characterization in the design of processing cores. In *IEEE International Symposium on Workload Characterization (IISWC)*. 75–85.

S. Parekh, S. Eggers, H. Levy, and J. Lo. 2000. *Thread-sensitive scheduling for SMT processors*. Technical Report UW-CSE-00-04-02. University of Washington.

M. Pinsky and S. Karlin. 2010. *An introduction to stochastic modeling*. Academic press.

S. E. Raasch and S. K. Reinhardt. 2003. The Impact of Resource Partitioning on SMT Processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 15–26.

Y. Sazeides and T. Juan. 2001. How to Compare the Performance of Two SMT Microarchitectures. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 180–183.

T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 45–57.

A. Snavely and D. M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2000), 234–244.

N. Tuck and D. M. Tullsen. 2003. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 26–34.

D. M. Tullsen and J. A. Brown. 2001. Handling Long-Latency Loads in a Simultaneous Multithreading Processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 318–327.

D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*. 191–202.

D. M. Tullsen, S. J. Eggers, and H. M. Levy. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In *International Symposium on Computer Architecture (ISCA)*. 392–403.

M. Van Biesbrouck, L. Eeckhout, and B. Calder. 2006. Considering all starting points for simultaneous multithreading simulation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 143–153.

M. Van Biesbrouck, L. Eeckhout, and B. Calder. 2007. Representative multiprogram workloads for multithreaded processor simulation. In *IEEE International Symposium on Workload Characterization (IISWC)*. 193–203.

M. Van Biesbrouck, T. Sherwood, and B. Calder. 2004. A co-phase matrix to guide simultaneous multithreading simulation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 45–56.

K. Van Craeynest and L. Eeckhout. 2011. The Multi-Program Performance Model : Debunking Current Practice in Multi-Core Simulation. In *IEEE International Symposium on Workload Characterization (IISWC)*. 26–37.

H. Vandierendonck and A. Seznec. 2011. Fairness Metrics for Multi-Threaded Processors. *Computer Architecture Letters* 10, 1 (Jan 2011), 4–7.

R. A. Velásquez, P. Michaud, and A. Seznec. 2013. Selecting benchmark combinations for the evaluation of multicore throughput. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 173–182.

J. Vera, F. Cazorla, A. Pajuelo, O. Santana, E. Fernandez, and M. Valero. 2007. FAME: Fairly measuring multithreaded architectures. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 305–316.